(54) ACCESS SYSTEM FOR SYSTEM CONTROL FILE OF ASYNCHRONOUS COMMUNICATION

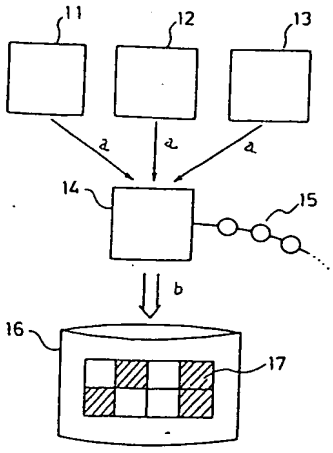
(11) 62-209635 (A) (43) 14.9.1987 (19) JP
(21) Appl. No. 61-24995 (22) 7.2.1986

(71) FUJITSU LTD (72) SHINKO NAKAGAWA

(51) Int. Cl⁴. G06F12/00,G06F9/46

PURPOSE: To omit the exclusive control between jobs by providing a control job for replacement of control files, subjecting queue control to the request issued from each job, and executing serially the processing updating a system control file of each request in terms of time.

CONSTITUTION: A control job 14 accepts the access requests supplied to a system control file 16 from the communication requester jobs 11~13 and connects these access requests to an access request queue 15 to perform successively the access processes of the file 16 at and after the head one. Then the job 14 carries out the replacement processing for acquisition of resources at the start of communication and for release of resources at the end of communication with all control blocks (shown by oblique lines) related to a single communication requester job that is taken out of the queue 15 in a control block 17 of the file 16. Then the part 14 takes the next communication requester job out of the queue and executes it.



a: request, b: access processing

BEST AVAILABLE COPY

A section from the book entitled "Distributed Operating System—what comes after UNIX", pg. 169-176, written by Mamoru Maekawa *et. al.*; date published: December 25, 1991; published by Kyoritsu Shuppan Kabushiki Kaisha.

9.4 Control of Simultaneous Execution

9.4.1 Common Clock

The basic problems of simultaneous execution control in a system lies in deciding and managing an order of phenomenon occurrences in the system. To solve these problems, one can consider to use a node which performs such management of the phenomenon occurrences (also called concentrated management). A lot of problems distinctive in a distributed system can be avoided by using the concentrated management method. However, this method has problems in aspects of efficiency and reliability. As regard to the distributed system, it is desirable to distribute a mechanism as well in implementing the simultaneous execution control management.

In the distributed system, a process request is generally transmitted in a message. A time taken in transmitting and receiving the message is variable. For example, when two processes A and B transmit messages to a process P in an order of A and B, the process P may receive the messages in an order of B and A, and so-called a revered order of arriving messages may occur. Under such circumstance, it is necessary for the processes A and B to have common clocks, so that the transmission times can be entered to their respective messages.

Rules for deciding the order of phenomenon occurrences in a system is as follows:

(1) If a phenomena *a* is executed before a phenomena *b* inside one process, then $a \rightarrow b$; and

(2) If the processes are communicating with one another, provided that transmission is *a* and reception is *b*, then $a \rightarrow b$.

In order to create the order relationship that meets this clock (i.e. if $a \rightarrow b$ then an occurrence time *a* < an occurrence time *b*), the followings are the case:

(1) each process (actually each node) has a counter, and the counter is increased every time a new phenomena occurs; and

(2) in cases of exchanging a message among a process 1 and a process 2, a value of the counter of the process 1 is attached to the message as a timestamp, and when a value of the timestamp is greater than

a value of counter of the process 2 that has received the message, then update the value of counter of the process 2 to be greater than the value of the timestamp.

If the values of timestamps for the phenomenon happen to be identical, then a process number (node number) can be used instead to assign the order of phenomenon occurrences. Virtually, the value of counter close to an actual time is adopted. In order to do so, each node in the system should have a sufficiently accurate clock as a counter, and when a message arrives at each node, the clock of each node itself is re-adjusted based on the transmission time of the arrived message and a minimum estimation of delayed time of the arrived message. According to the result by Lamport [Lamport 78], all the clocks inside the system can be adjusted at an error below $d\ (2k\tau + \xi)$ provided that an error of the clock is k, a diameter of the network is d (set of nodes that cannot communicate without passing through a vicinity of d nodes), a maximum interval of message transmission is τ, and a message delayed time is ξ which cannot be predicted.

9.4.2 Problems in exclusive control

When the exclusive control in the distributed system is performed by the concentrated management method, only two messages should be sufficient in order to perform a single exclusive control, namely: a request message being transmitted from a request process to a management process; and a response message of the request message. However, the concentration management method has problems as described in the previous section, and it is also ideal to distribute the exclusive control management. The distributed management method requires the following properties:

(1) each node has the same amount of management information;
(2) each node performs decision using an identical algorithm;
(3) each node allots loads equally;
(4) cost involved in initializing the control from each node is all same; and
(5) a fault in a single node does not lead to a termination of the whole system.

Also, assumption and characteristics common to the algorithms which will be described below are that:

(1) a single process is contained in each node;

(2) messages are processed in the order of their transmission times;
(3) messages are correctly received within a finite time; and
(4) a network is fully closed.

As a basic solution to the problems, there is an algorithm of Lamport [Lamport 78]. This algorithm adopts the previously described timestamp.

(1) a process 1 requesting for a resource enters a request into a queue and transmits a REQUEST message to all other processes;
(2) a process which has received the REQUEST message enters the request of the process 1 into the queue and returns a REPLY message to the process 1;
(3) when the request of the process 1 is leading in the queue, and when the REPLY messages are received from all other processes that are newer than the request of the process 1, then the request of the process 1 can access the resource;
(4) when the process 1 is releasing the resource, the process 1 removes its request from the queue, and transmits a RELEASE message to all other processes; and
(5) a process which has received the RELEASE message removes the request of the process 1 from the queue.

Using this method, 3X(N-1) messages are required for a single exclusive control. N indicates a number of all the processes.

As opposed to this method, according to the Ricart and Agrawala's algorithm [Ricart 81]:

(1) a process 1 requesting for a resource transmits a REQUEST message to all other processes;
(2) a process which has received the REQUEST message returns a REPLY message only if the required REQUEST message of the process 1 is older than a request of the process or if the process is not requesting at all. The REPLY message is not transmitted if the REQUEST message of the process 1 is not older than the request of the process;
(3) the process 1 which has transmitted the request can access the resource when the REPLY messages from all other processes are received; and
(4) when the process 1 is releasing the resource, if there are REPLY messages not being transmitted at (2), they are transmitted.

Under this method, 2X(N-1) messages are required for a single

exclusive control.

In order to access the resource from each process using the two methods described above, a permission of "all consistent" from all other processes must be obtained. By adopting a majority voting method of Thomas', this condition of the "all consistent" is loosened so that the permissions need be obtained from only more than half of the process. By adopting the Thomas' method, the numbers of processes and messages required for a single exclusive control can be halved. By adopting such voting method having a weight on it, only more than half of votes held by all the processes has to be obtained, and not more than half of the processes.

Maekawa's algorithm only requires a much lesser number of messages [Maekawa 85].

Let's suppose that groups of processes including processes i and j are written as Si and Sj, respectively. The following conditions are required in order for the processes i and j ($1 \leqq i, j \leqq N$) to access the resource. Their request messages are transmitted and their respective reply messages to return to the other group, respectively.

(1) $\forall i\ j\ \ Si \cap Sj \neq \phi$

(2) $|S_1| = |S_2| = \cdots = |S_N| = K$ (constant)

(3) $\forall i\ |\{j \mid Sj \ni i\}| = D$ (constant)

Further, provided that

(4) $\forall i\ i \ni Si$

then an actual number of messages transmitted can be reduced. Under these condition, K can be set approximately to $\sqrt{N}$. This means that Maekawa's algorithm must communicate with at least $\sqrt{N}$ processes for implementing the exclusive control. When calculate Si to meet the conditions (1) to (4), then the exclusive control is implemented at $O(\sqrt{N})$ messages. Refer to a paper by Maekawa 85 on how to calculate Si. The exclusive control is performed in the following order.

(1) a process i requesting for a resource transmits a REQUEST message to all the members of Si.

(2) Processes of Si that have received the REQUEST message of the process i is locked by i if not yet locked by any other processes, and the locked process returns a LOCKED message to i. If a process was already being locked by other process, then the REQUEST message of i enters a queue, however, if either one of a REQUEST message of

currently locked process or the other REQUEST messages that may be present in the queue is older than the REQUEST message of *i*, then a FAILED message is returned to *i*. Otherwise, an INQUIRE message is sent to a process currently locking this process, and inquires whether the process has obtained all the necessary locks.

(3) The process that has received the INQUIRE message returns a RELIQUISH message, provided that the process has already received the FAILED message.

(4) A process that has received the RELIQUISH message is released from a current lock, and the REQUEST message of *i* re-enters the queue. Next, this process is locked by the oldest REQUEST message in the queue, and a LOCKED message is returned to the process that sent the oldest REQUEST message.

(5) The process *i* can access the resource when it has received the LOCKED messages from all members of *Si*

(6) When the process *i* is releasing the resource, send a RELEASE message to all members of *Si*.

(7) a process that has received the RELEASE message is released from the current lock, and is locked by the most oldest REQUEST message in the queue. A LOCKED message is returned to the process that sent the oldest REQUEST message.

Using this method, $C\sqrt{N}$ messages (where C=3 to 5) are required for a single exclusive control.

As regard to the previously described algorithms, none the processes has an exclusive rights (tokens) to access the resources. An algorithm having the least number of messages is created by loosening the conditions and by exchanging the tokens to access the resources. For example, there are algorithm having N messages required for exclusive control [Suzuki 85] and algorithm having log N messages for communication between the processes by configuring a tree-like topology [Trehel 86].

9.4.3 Concurrency control

When a plurality of transactions are executed concurrently, there are possibility that the read and write operations defined in the transactions are executed in various orders. Especially, when reading and writing the same data item in the same file, a result of the concurrent execution cannot be predicted. In order to prevent this from happening,

even though a plurality of transactions are executed concurrently, the transactions must be controlled in some way so that the result of concurrent execution will be same as a result of successive execution. Such management of executing transactions is known as a serialization, and its control method is known as a concurrency control. Examples of the result of concurrency control and the result of no concurrency control are illustrated in Fig.9.7.

The concurrency method is largely classified into three types.

(1) method based on lock

(2) method based on timestamp

(3) optimistic method

These methods are described below.

When a transaction 1 is executing a series of operations to a certain resource, an inconsistency will arise if another transaction executes the same resource while being executed by the transaction 1, because this resource is in an inconsistent state. To avoid the transactions sharing the resource under the inconsistent state, a mutual exclusion is performed when the transaction accesses the resource in order to get around such inconvenience.

If all the transactions are well-formed, and are two-phase, they can be serialized [Eswaran 76]. For the transactions to be well-formed, the following three properties must be met by the transactions:

(1) a resource is locked before accessing it

(2) a resource that is already being locked will not be locked again

(3) all locks are unlocked before completing the process

For the transactions to be 2-phase, once the resource is unlocked during the transaction, there will be no more locking of the resource after that. In such cases, a process is divided into two phases of lock and unlock as illustrated in Fig.9.8. Such locking method is known as two-phase locking. The method based on lock is known to be more effective than other methods, and it is quite commonly being utilized. However, there is a possibility of dead lock occurring, which must be detected and avoided. The dead lock will be described in the next section.

Practically, the question of vital importance in a lock is a granularity of the lock. For example, let's consider locking a file, then if a single file is set as a concurrent access unit, then the concurrency will be decreased prominently. Locking of a logic record

of the file is more common.

Also, in order to increase the concurrency, there are cases of inducing two types of lock for read and write. We refer to a lock for reading as a read lock, and a lock for writing as a write lock. A transaction that only performs reading to a particular data item will not compete with another transaction that only performs reading to the same data item. In such cases, a plurality of transactions shares the read lock. However, access by another transaction is prohibited for those data that are locked by the write lock. Table 9.2 illustrates the lock rule.

Read is more frequently performed compared to write, therefore, this method seems to give an improved effect on the concurrency, even though a transaction which performs read only will completely interfere a progress of transaction that accompanies write. Gifford is proposing an induction of "intention-to-write lock" [Gifford 79]. Using this method, a commit lock is used instead of the write lock. Table 9.3 illustrates the lock rule of "intention-to-write lock". When the write is being performed, a data is locked by the intention-to-write lock, and further, when the transaction performs commit, the intention-to-write lock is changed to the commit lock. Under a locked condition by the intention-to-write lock, the read is being executed concurrently.

With regard to the method based on the timestamp, the timestamps are provided to all the transactions beforehand so that this method can serialize the transactions by only permitting access in an order of the timestamps [Bernstein 81] when requests for accessing to the resource are competing. Its fundamental objectives are to make a write valid only after performing read and write of the previous transaction, and to make a read valid only after performing write of the previous transaction.

For the method based on the timestamps, the timestamp is assigned to the transaction at a time when the timestamp is being created. This timestamp is always attached to an access request of the data item. Also, to each data item, a timestamp Tw of a transaction which performed the write last, and a timestamp Tr of a transaction which performed the read last are recorded. Figure 9.9 illustrates a state of concurrency control based on the timestamps. The read and write are processed as follows.

(1) Suppose that a request to read the timestamp T is made, T<Tw, if a transaction newer than a transaction which has already

performed write, the transaction which requested it is aborted. If $T \geqq Tw$, then the read is performed, and Tr is updated to max (Tr, T).

(2) Suppose that a request to write the timestamp T is made, $T < \max(Tr, Tw)$, if a transaction newer than a transaction has already performed either the read or write, the transaction which requested it is aborted. If $T \geqq \max(Tr, Tw)$, then the write is performed, and Tw is updated to max (Tw, T)

When the transaction is aborted, all the updating performed by the transaction are deleted, and an execution is re-attempted by providing a new timestamp to the transaction. This is known as a callback. When the callback frequently occurs, no doubt that an efficiency declines prominently.

The write rule of Thomas attempts to improve the efficiency by performing the followings to the write (item 2 above). When a write request of the timestamp T is small, $Tw < T < Tr$, or in other words, a transaction 1 newer than a transaction 2 has already read a result which is older than what the transaction 1 is trying to re-write, the transaction 1 that performed the write request is aborted. Nothing happens if $T < Tw$. For the other cases, write is performed, and Tw is updated to max(Tw, T). This method decreases a possibility of aborting by competition between writes by ignoring those write requests which are erased due to repeated writing before any one of the transactions reads it.

Further, a method known as a multi-vision is used to avoid abort at reading. In this method, write is not something that re-writes an original value, but it has a meaning to record a set of the timestamp Tw for the write transaction and a content of the writing V (Tw, V) as a new version.

(1) Suppose that a read request of the timestamp T is made, then a content of the most recent version from versions having the timestamp Tw smaller than T is read out.

(2) Suppose that a write request of the timestamp T is made, if a transaction having a large timestamp has already read the most recent version $T > Tw$, this write is rejected, and the transaction is aborted. For the cases other than this, a new version is created.

Conservative timestamp ordering is a method which does not need to reactivate the transaction by aborting. This method first of all collects the requests from all the transactions, then processes the most oldest request (i.e. request having the minimum timestamp). This prevents a later arrival of a request from a transaction, that might be carrying an older timestamp than a timestamp of the executed request. Therefore, aborting of the transaction does not occur. However, this method has a problem that a non-competing request will also be scheduled for execution according to the order of timestamps. To solve this problem, one can analyze the competing transactions beforehand and apply this method after having classified the competing transactions.

To solve the problems of the method based on the lock and the method based on the timestamp, Kung and Robinson have proposed the optimistic method [Kung 81]. This is an idea that for the two processes being executed among transactions, if an access competition for the same data item are known as not occurring frequently, then there is no need to look up the competition for every read or write. Let the transaction to execute concurrently, and check to see whether the competition has actually occurred or not by analyzing the group of data items where the read and write is performed after completing the transaction. If the access competition is occurring, then the transaction is aborted. A copy of the data is stored for every transactions, and an actual operation is performed on that copy. This operation cannot be observed from other transactions.

9.4.4 Solution to dead lock

This section describes a method of solving the dead lock which is a problem of concurrency control based particularly on a lock.

In general, there are two measures against the dead lock, namely: a prevention and a detection of the dead lock. The most simple method of preventing the dead lock is to announce all the resources that will be used by a transaction, and lock them all up upon commencing the transaction. The method is ideal when occupying time is short, however, a rate of resource usage may change for the worse, and may have a danger of inviting a starved condition. There is also a method of assigning the order according to the types of resources, although a decline in the rate of resource usage is unavoidable.

So-called "wait-for graph" is commonly used in detecting the dead

lock. This is a graph created by defining a direction indicator from a transaction B to a transaction A when the transaction B is waiting for the resources locked by the transaction A to unlock them. A fact that the closed path of the direction indicator is present in this graph becomes an essence to the dead lock occurrence. After detecting the dead lock, the graph must decide which transaction should be aborted.

There are the following methods in detecting the closed path.

(1) when each transaction attempts to access the resource, the dead lock detection message is transmitted in accordance to the direction indicator coming from the transaction, and check to see if this message will return to the original transaction.

(2) each transaction stores all the transactions that can be reached along the direction indicator (arriving possible group). When the transaction A attempts to access the resource locked by the transaction B, the transaction A confirms whether or not the transaction A is stored in the arriving possible group stored by the transaction B.

The method of (1) is not efficient since the transaction must be checked by successively transmitting the messages. The method of (2) has a problem that changing of the arriving possible group at unlocking time is not easy.

As another measure against the dead lock, there is a method of using a lock timeout. This is the method of attaching a time limit to the lock, and releasing the lock when the access requests from the others are received. This method is commonly used in a file server. The problem with this method is the difficulty in setting the time limit because the execution time of transaction changes depending on the load.

Figure 9.7 Examples of Concurrency Control

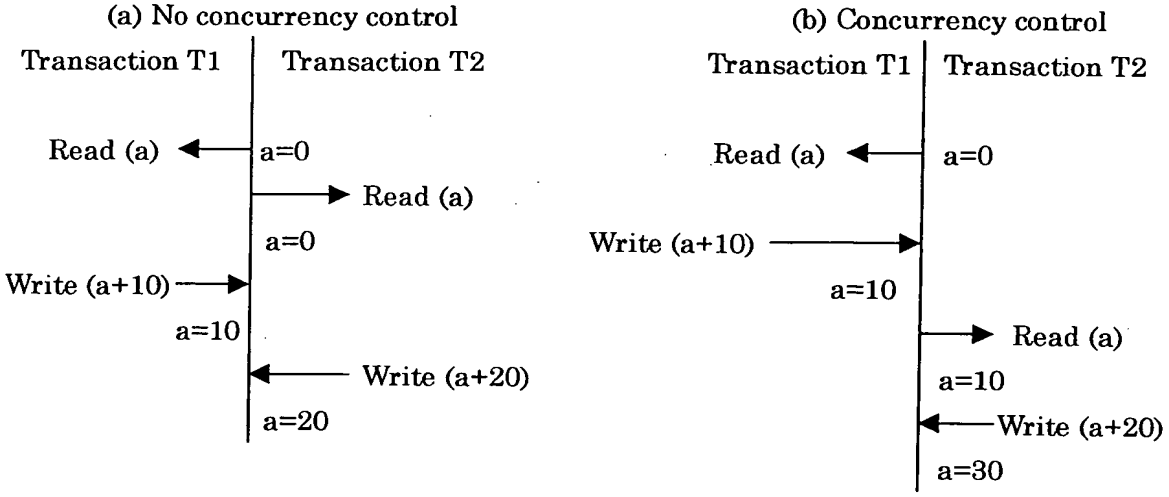


Figure 9.8 Two-phase locking

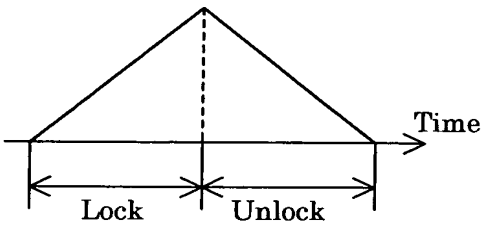


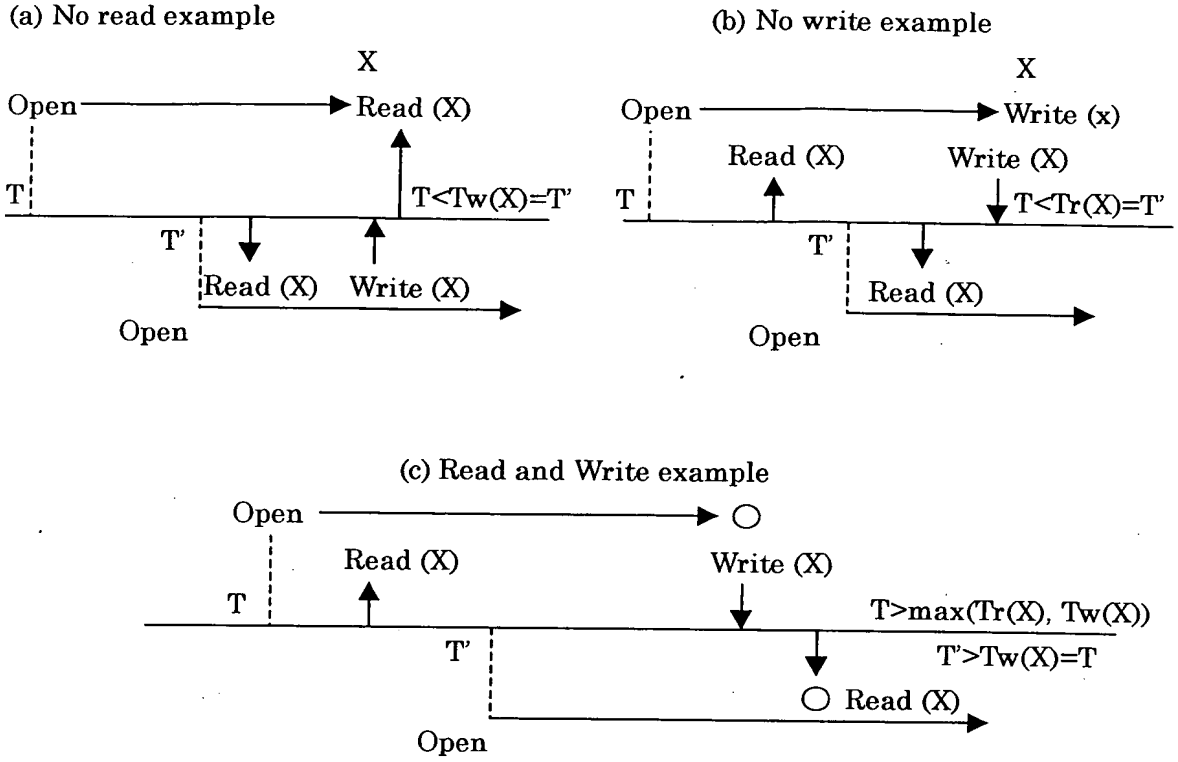
Table 9.2 Read lock and Write lock

Request lock \ Locked status of Object	No lock	Read	Write
Read	Yes	Yes	No
Write	Yes	No	No

Table 9.3 Lock rule inducing intention-to-write lock

Request lock \ Locked status of Object	No lock	Read	Intention-to-Write	Commit
Read	Yes	Yes	Yes	Wait
Intention-to-write	Yes	Yes	Wait	Wait
Commit	Yes	Wait	Wait	Wait

Figure 9.9 Timestamp method



分散オペレーティングシステム
UNIXの次にくるもの

前川 守・所 真理雄・清水謙多郎 編

Distributed Operating System - What comes after UNIX
by Mamoru Maekawa, et al.

共立出版株式会社

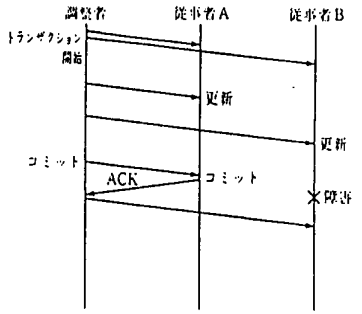


図9.5 1相コミットメント

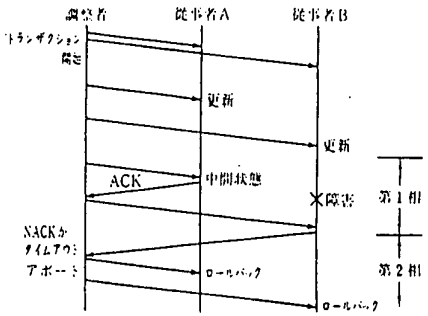


図9.6 2相コミットメント

者はすべての従事者にコミットの用意があるか問い合わせる．各従事者は，返答を調整者に返すと同時に，自らアボートする権利を放棄し，コミットもアボートもできる中間状態に移行する．調停者は，全従事者から返答を受け，これがすべて肯定応答ならコミット，そうでなければアボートを全従事者に指示する．これが第2相である．これにより，関係する全ノードが同一の決定を下すことになる．2相コミットメントには，メッセージの転送順序をあらかじめ決めておくことにより，調整者を必要としない分散型もある．

9.3.4 オペレーティングシステムにおけるトランザクション

分散オペレーティングシステムでよく用いられるトランザクションの形態には次のようなものがある．

(1) 入れ子型トランザクション (nested transaction)

一つのトランザクションの実行の中に，いくつかのトランザクション（サブトランザクション）の実行が含まれているトランザクションをいう．トランザクションの実行はすべてのサブトランザクションが完了しないと終了しない．サブトランザクションがさらに入れ子になっていてもよい．この型のトランザクションは，途中障害が発生しても親トランザクションを中断せずに関連するサブトランザクションだけ再試行すればよいという利点がある．LOCUS [Mueller 83] や Argus [Liskov 88] で実現されている．

(2) 長期トランザクション (long-term transaction)

分散プログラミング環境においては，トランザクションの実行が長くなりがちである．これは，並行性を低くすることになり，性能の低下を招くことがある．

(3) ユーザ定義オブジェクトに対する操作

オブジェクトの型や操作のようなセマンティクス情報を用いて，独立した操作を並行して実行できるようになる．上記のようなトランザクションにおいて並行性を上げることができる．

オブジェクトの型や操作などのセマンティクスを用いた並行性制御については，後で述べる．

9.4 同時実行の制御†

9.4.1 共通の時計

システム内で生起した事象の生起順序を決定し，管理することは，同時実行の制御を行なう上での基本的な問題である．この管理を一つのノードが集中して行なう（集中管理）ことも考えられる．こうすれば，分散システム特有の問題の多くを回避することができる．しかしこの方式は，効率，信頼性の両面で問題がある．分散システムにおいては，同時実行性制御管理のための機構もまた分散されて実現されることが望ましい．

分散システムでは，一般にプロセスに対する要求はメッセージによって伝達される．メッセージが送信されてから受信されるまでの時間はまちまちである．したがって，たとえば二つのプロセスA，BがプロセスPにA，Bの順でメッセージを送信しても，PにはB，Aの順に到着し，いわゆる到着順序の逆転が起こることがある．この場合，Pではメッセージが送信された順（A，Bの順）で処理したいという場合も多い．このためには，A，Bが共通の時計をもって，それぞれのメッセージの中にその送信時刻を記入しておく必要がある．

システム内の事象の生起順序を決定する規則は次のとおりである．

（1） 一つのプロセスで事象 a が事象 b よりも先に実行されるとき，$a \rightarrow b$ である．

（2） プロセス間で通信する場合，送信を a，受信を b とするならば，$a \rightarrow b$ である．

この二つの規則に，推移律（$a \rightarrow b$，$b \rightarrow c$ ならば $a \rightarrow c$）を加えることによって，システム内の事象に（半）順序関係を導入することができる．

この順序関係を満たす時計（$a \rightarrow b$ ならば，a の生起時刻$<b$ の生起時刻）を作るには，以下のようにすればよい．

（1） 各プロセス（実際にはノード）はカウンタをもち，新しい事象が起こるたびにそのカウンタを増加させる．

（2） メッセージをやりとりする場合は，そのカウンタの値をタイムスタンプとしてメッセージに付加し，受信したプロセスは，そのタイムスタンプが自分のカウンタよりも大きければ，自分のカウンタをタイムスタンプより大きくなるように変更する．

同じタイムスタンプをもつ事象については，プロセス（ノード）番号を使うことによって順序を付けることができる．実際には，実時刻に近いカウンタの値が用いられる．そのためには，システム内の各ノードは十分に正確な時計をカウンタとしてもち，メッセージが到着したときには，その送信時刻と伝送遅れ時間（の最小）の見積りを用いて自分の時計を合わせ直すようにすればよい．Lamportの結果[Lamport 78]によると，時計の誤差を k，ネットワークの直径を d（d 個の辺を通らないと通信できないノードの組が存在する），メッセージ送信の最大間隔 τ，予測不可能なメッセージ遅れ時間 ξ に対し，おおよそ $d(2k\tau+\xi)$ 以下の誤差でシステム内のすべての時計を合わせることができる．

9.4.2 排他制御問題

分散処理システムにおける排他制御を集中管理方式で行なう場合，1回の排他制御を行なうのに，要求プロセスから管理プロセスへの要求メッセージと，それに対する応答メッセージの二つだけで十分である．しかし，この方式は前節で述べたように問題があり，排他制御管理も分散して行なうのが望ましい．分散管理方式では

（1） 各ノードは同量の管理情報をもつ．

（2） 各ノードは同じアルゴリズムで決定を行なう．

（3） 各ノードは等しく負荷を分担する．

（4） 各ノードから制御を起動するコストが同じである．

（5） 単一ノードの故障が全システムの停止につながらない．

という性質が必要である．また，以下に示すアルゴリズムに共通する仮定と特徴は次のとおりである．

（1） 各ノードには一つのプロセスが存在する．

（2） メッセージは送信された順序で処理される．

（3） メッセージは有限の時間で正しく着信する．

（4） ネットワークは完全結合である．

この問題の基本的な解法としてLamportのアルゴリズム[Lamport 78]がある．

これは，先に述べたタイムスタンプを用いるものである．

（1） 資源を要求するプロセスは，要求をキューに入れ，ほかのすべてのプロセスにREQUESTメッセージを送る．

（2） REQUESTメッセージを受け取ったプロセスは，その要求をキューに入れ，REPLYメッセージを返す．

（3） 自身の要求が自身のキューの先頭にあり，他のすべてのプロセスからその要求よりも新しいREPLYメッセージを受け取っているとき，資源にアクセスすることができる．

（4） 資源を解放するとき，プロセスはその要求をキューからはずし，他のすべてのプロセスにRELEASEメッセージを送る．

（5） RELEASEメッセージを受けたプロセスは，その要求をキューからはずす．

この方式では，1回の排他制御につき，$3\times(N-1)$ 個のメッセージを必要とする．ここに，N は全プロセス数を示す．

これに対して，RicartとAgrawalaのアルゴリズム[Ricart 81]では

（1） 資源を要求するプロセスは，他のすべてのプロセスにREQUESTメッセージを送る．

（2） REQUESTメッセージを受けたプロセスは，自身が要求を出していないか自身の要求よりもこの要求メッセージのほうが古いとき，REPLYメッセージを返す．そうでないときは，REPLYメッセージの送信を止めておく．

（3） 要求を出したプロセスは，REPLYメッセージを他のすべてのプロセスから受け取ったとき，資源にアクセスすることができる．

（4） 資源を解放するとき，(2)で止めてあるREPLYメッセージがあればこれを送信する．

この方式では，1回の排他制御につき，$2\times(N-1)$ 個のメッセージを必要とする．

上に挙げた二つの方法では，各プロセスは資源をアクセスするのに，他のすべてのプロセスから「満場一致」の許可を得なければならない．Thomasの多数決方式を適用し，この「満場一致」の条件をゆるめ，過半数のプロセスから許可を得ればよいようにすることによって，排他制御1回に要するプロ

† 本節の内容は，文献[前川 87]に加筆したものである．

セス数およびメッセージ数を半減することができる．重み付き投票方式を適用すれば，プロセスの過半数ではなく，全プロセスのもつ票数の過半数を得ればよいことになる．

前川のアルゴリズム［Maekawa 85］は，さらに少数のメッセージしか必要としない．

いま，プロセス i，j $(1 \leqq i, j \leqq N)$ が資源にアクセスするために要求メッセージを送り，それに対する応答メッセージが返される相手プロセスの集合をそれぞれ，S_i，S_jと書くことにすると，

(1) $\forall i, j$ について $S_i \cap S_j \neq \emptyset$

(2) $|S_1| = |S_2| = \cdots = |S_N| = K$（ある定数）

(3) $\forall i \, |\{j \mid S_j \ni i\}| = D$（ある定数）

であることが必要である．さらに，

(4) $\forall i \; i \in S_i$

とすれば，実際に送られるメッセージ数を少なくすることができる．これらの条件のもとで，K をほぼ $\sqrt{N}$ にすることができる．すなわち，少なくとも $\sqrt{N}$ 個のプロセスと通信しなければ排他制御は実現せず，また，上の(1)～(4)を満たすように S_iをとれば，$O(\sqrt{N})$のメッセージで実現できることになる．S_iの求め方については文献［Maekawa 85］を参照されたい．排他制御は次のような手順で行なわれる．

(1) 資源を要求するプロセス i は S_i のすべてのメンバに REQUEST メッセージを送る．

(2) プロセス i の REQUEST メッセージを受けたプロセスは，他のプロセスによってまだロックされていなければ i によってロックされ，LOCKED メッセージを i に返す．すでにロックされている場合は，REQUEST メッセージはキューに入れられるが，現在このプロセスをロックしている REQUEST メッセージかキューにある他の REQUEST メッセージのほうが i の REQUEST メッセージより古いものであれば，FAILED メッセージを i に返す．そうでなければ，現在このプロセスをロックしているプロセスに対し INQUIRE メッセージを送り，そのプロセスが必要なすべてのロックを得たかどうか問い合わせる．

(3) INQUIRE メッセージを受けたプロセスは，すでに FAILED メッセージを受け取っていれば RELIQUISH メッセージを返す．

(4) RELINQUISH メッセージを受けたプロセスは現在のロックから解放され，この REQUEST メッセージは再びキューに入れられる．次にこのプロセスはキューの中で最も古い REQUEST メッセージにロックされ，REQUEST メッセージを出したプロセスに LOCKED メッセージが返される．

(5) S_i のすべてのメンバから LOCKED メッセージを受け取ったら，プロセス i は資源にアクセスすることができる．

(6) プロセス i が資源を解放するとき，S_i のすべてのメンバに RELEASE メッセージを送る．

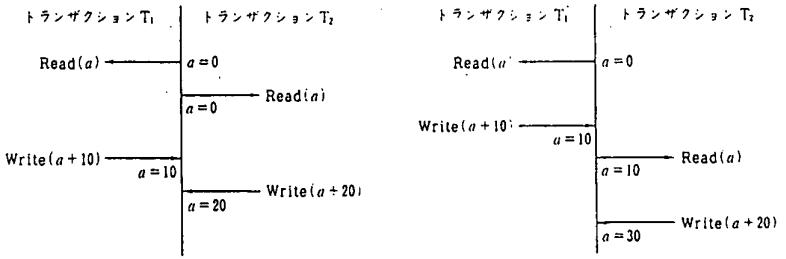
(7) RELEASE メッセージを受けたプロセスは，現在のロックから解放され，キューの中で最も古い REQUEST メッセージにロックされる．その REQUEST メッセージを出したプロセスに対して LOCKED メッセージが返される．

この方式では，1回の排他制御につき，$C\sqrt{N}$ (C は3から5) 個のメッセージを必要とする．

以上のアルゴリズムは，資源にアクセスする際にどのプロセスも専有権をもたない．この条件をゆるめ，資源に対する専有権（トークン）をやりとりすることで，よりメッセージ数が少ないアルゴリズムを作ることができる．たとえば，排他制御に必要なメッセージ数を N 個としたアルゴリズム［Suzuki 85］やプロセス間通信のトポロジを木状にすることによって log N 個としたアルゴリズム［Trehel 86］がある．

9.4.3 並行性制御

複数のトランザクションが並行して実行されるとき，それらの中に定義された読出しおよび書込み操作はさまざまな順序で実行される可能性がある．とくに，同じファイルの同じデータ項目に対する読み書きがある場合は，その結果を予想することはできない．こうしたことを防ぐには，複数のトランザクションが実際には並行して実行されても，その結果はそれらが逐次的に実行されたのと同じになるよう



(a) 並行性制御を行なわない場合　(b) 並行性制御を行なった場合

図 9.7 並行性制御の例

に制御しなければならない．このようなトランザクションの実行の管理を直列化 (serialization) といい，その制御方式を並行性制御 (concurrency control) という．図9.7に並行性制御を行なったときと行なわなかったときとで実行結果が異なる例を示す．

並行性制御の方式には，大きく分けて次の三つの方式がある．

(1) ロックに基づく方法

(2) タイムスタンプに基づく方法

(3) 楽観的 (optimistic) 方法

以下，これらの方法について説明する．

トランザクションがある資源に対して一連の操作を実行している場合，その途中の矛盾した状態を用いて他のトランザクションが実行を行なうと矛盾が生じる．そこで，途中の一貫性のない状態では他のトランザクションと資源を共用しないように，資源アクセスに対して相互排除を行なえば不都合を回避することができる．

すべてのトランザクションが整形 (well-formed) で2相 (two-phase) であれば，それらを直列化できることが知られている［Eswaran 76］．トランザクションが整形であるとは，次の三つの性質を満たすことをいう．

(1) 資源にアクセスする前にはそれをロックする．

(2) すでにロックされている資源はロックしない．

(3) 処理を完了する前にすべてのロックを解除する．

2相であるとは，トランザクション中で一度ロック解除（アンロック）を行なったらそれ以降はロッ

クを行なわないことをいう．この場合，図9.8に示すように，処理がロックとアンロックの二つの相に分割されることになる．こうしたロッキング方式を2相ロッキング（two-phase locking）という．ロックに基づく方法は，他の方法に比べて効率的であることが知られており，実際によく用いられている．ただし，デッドロックを発生する可能性があり，その検出や回避が必要である．デッドロックについては次節で説明する．

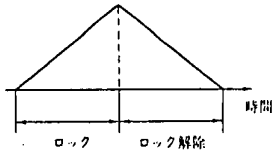


図9.8 2相ロッキング

ロックにおいて実際上重要な問題として，ロックの粒度（granularity）がある．たとえば，ファイルに対するロックを考えたとき，一つのファイルを並行アクセスの単位としてしまうと，並行性を著しく下げてしまうことになる．ファイルの論理レコードをロックの対象にすることが多い．

また，並行性を上げるため，読出しと書込みで2種類のロックを導入することがある．読出しに対するロックを読出しロック（read lock），書込みに対するロックを書込みロック（write lock）という．あるデータ項目に対して読出ししか行なわないトランザクションは，同じデータ項目に対してやはり読出ししか行なわないトランザクションと決して競合することがない．この場合，複数のトランザクションが読出しロックを共有することになる．ただし，書込みロックされているデータについては，他のトランザクションからのアクセスが禁止される．表9.2はこうしたロックの規則を示したものである．

表9.2 読出しロックと書込みロック

要求ロック ＼ オブジェクトのロック状態	ロックなし	読出し	書込み
読出し	可	可	不可
書込み	可	不可	不可

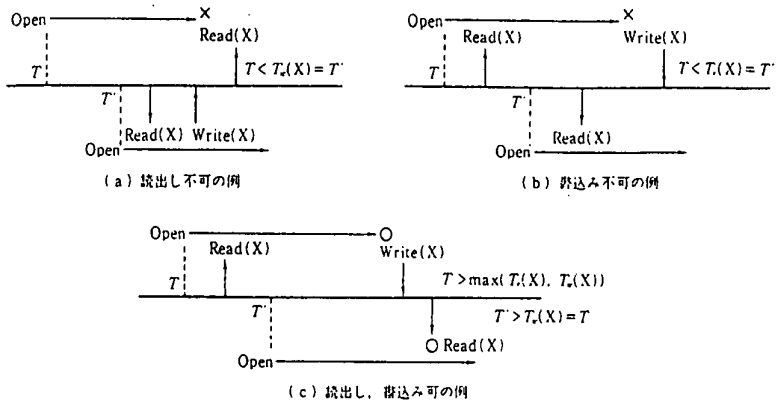
表9.3 書込み意志ロックを導入したロック規則

要求ロック ＼ オブジェクトのロック状態	ロックなし	読出し	書込み意志	コミット
読出し	可	可	可	待ち
書込み意志	可	可	待ち	待ち
コミット	可	待ち	待ち	待ち

読出しは書込みに比べて頻繁に行なわれるので，この方法は並行性の向上に効果があると考えられるが，それでも読出ししか行なわないトランザクションが書込みを伴うトランザクションの進行を完全に妨げてしまうことになる．Gifford はこれに対して，書込み意志ロック（intention-to-write lock）の導入を提案している［Gifford 79］．この方式では，書込みロックの代わりにコミットロックが用いられる．表9.3にロック規則を示す．書込みが行なわれるときには書込み意志ロックでロックされ，さらにトランザクションがコミットするときこれがコミットロックに変更される．書込み意志ロックでロックされた状態では，読出しを並行して実行することができる．

タイムスタンプに基づく方式は，すべてのトランザクションにあらかじめタイムスタンプを与え，資源に対して競合するアクセスを要求した場合にはそのタイムスタンプの順でしかアクセスを許さないことでトランザクションを直列化しようというものである［Bernstein 81］．その基本的な方針は，書込みについては以前のトランザクションの読出しおよび書込みが行なわれた後でのみ有効とし，読出しについては以前のトランザクションの書込みが行なわれた後でのみ有効とするものである．

タイムスタンプに基づく方式では，トランザクションはその生成時にタイムスタンプが割り当てられる．データ項目に対するアクセス要求には，必ずこのタイムスタンプが付加される．また，各データ項目には，それに最後に書込みを行なったトランザクションのタイムスタンプ T_w と，最後に読出しを行なったトランザクションのタイムスタンプ T_r が記録される．図9.9にタイムスタンプに基づく並行性



（a）読出し不可の例
（b）書込み不可の例
（c）読出し，書込み可の例

図9.9 タイムスタンプ方式

制御の様子を示す．読出しと書込みは次のように処理される．

（1） タイムスタンプ T の読出し要求があった場合，$T<T_w$，すなわちそのトランザクションよりも新しいトランザクションがすでに書込みを行なっていれば，この要求を行なったトランザクションはアボートされる．$T \geq T_w$ ならば読出しが行なわれ，T_r は $\max(T_r, T)$ に更新される．

（2） タイムスタンプ T の書込み要求があった場合，$T<\max(T_r, T_w)$，すなわちそのトランザクションよりも新しいトランザクションがすでに読出しか書込みを行なっていれば，この要求を行なったトランザクションはアボートされる．$T \geq \max(T_r, T_w)$ ならば書込みが行なわれ，T_w は $\max(T_w, T)$ に更新される．

トランザクションがアボートされた場合，そのトランザクションが行なった変更のすべてが取り消され，トランザクションには新しいタイムスタンプが与えられて再度実行が試みられる．これをコールバック（callback）という．コールバックが頻繁に起こると効率は著しく低下する．

Thomas の書込み規則は，書込み（上記(2)）に関して次のような工夫を行ない，効率を上げようというものである．タイムスタンプ T の書込み要求があったとき，$T_w<T<T_r$，すなわちそのトランザクションよりも新しいトランザクションが，そのトランザクションが書き換えようとするよりも古い結果をすでに読み出していれば，この要求を行なったトランザクションはアボートされる．$T<T_w$ ならば何もしない．それ以外の場合には書込みが行なわれ，T_w は $\max(T_w, T)$ に更新される．この方法では，どのトランザクションも読み出さないうちに重ね書きされて消されてしまうような書込み要求を無視することによって，書込みどうしの競合によるアボートの可能性を減らしている．

さらに，読出し時のアボートを回避するための方式として，マルチバージョン方式（multiversion）がある．この方式では，書込みは，もとの値を書き換えるのではなく，書き込んだトランザクションのタイムスタンプ T_w と書き込んだ内容 V の組 (T_w, V) を新しいバージョンとして記録することを意味する．

（1） タイムスタンプ T の読出し要求があった場合，T よりも小さいタイムスタンプ T_w をもつバージョンのうち最新のバージョンの内容が読み出される．

(2) タイムスタンプ T の書込み要求があった場合，T よりも大きなタイムスタンプをもつトランザクションが，$T > T_w$ である最新のバージョンをすでに読み出していれば，この書込みは拒否され，トランザクションはアボートされる．それ以外の場合は，新しいバージョンが作成される．

保守的タイムスタンプ方式 (conservative timestamp ordering) は，トランザクションをアボートして再起動しなくてもよいようにするための方式である．この方式では，すべてのトランザクションからの要求をまず集め，そのうえで，最も古い（タイムスタンプが最小の）要求を処理する．これによって，より古いタイムスタンプをもつトランザクションからの要求が，すでに実行してしまった要求よりも後から到着することを防いでいる．したがって，トランザクションのアボートは起こらない．しかし，この方式では，競合しない要求もつねにタイムスタンプの順で実行するようスケジュールされてしまうという問題がある．この問題を解決ためには，競合しうるトランザクションをあらかじめ解析し，トランザクションを分類した後に，競合するものだけこの方式を適用することが考えられる．

ロックに基づく方式やタイムスタンプに基づく方式の問題を解決するため，Kung と Robinson は楽観的な方法を提案した [Kung 81]．これは，二つのプロセスが実行するトランザクションの間で，同じデータ項目に対するアクセス競合がめったに起こらないことがわかっていれば，読出しあるいは書込みごとに競合を調べる必要がないという発想である．トランザクションを並行して実行させておき，トランザクション終了時に読出しおよび書込みが行なわれたデータ項目の集合を解析して，実際に競合があったかどうかを調べる．もし競合があればトランザクションはアボートされる．トランザクションごとにデータのコピーを保持し，それに対して実際の操作が行なわれる．これらは，他のトランザクションからは見えないようになっている．

9.4.4 デッドロックの解決

本節では，とくにロックに基づく並行性制御問題のデッドロックの解決法について説明する．

一般に，デッドロックの対策には防止と検出の二つがある．デッドロック防止の最も簡単な方法は，トランザクションが用いる資源をすべて宣言しておいて，それらをトランザクション開始時にすべてロックしてしまうというものである．この方法は，占有時間が短い場合はよいが，資源利用率を悪化させ，また飢餓状態を招くおそれがある．資源の種別に応じて割当て順序を適用する方法もあるが，資源利用率の低下は免れない．

デッドロックの検出には wait-for グラフ (wait-for graph) がよく用いられる．これは，トランザクションAがロックしている資源がアンロックされるのをトランザクションBが待っているとき，BからAに向かう有向枝を定義することによってできるグラフである．このグラフに有向枝の閉路が存在することがデッドロック発生の必要十分条件になる．デッドロックが検出された後は，どのトランザクションをアボートするか決定しなければならない．

閉路の検出には次のような方法がある．

(1) 各トランザクションが資源にアクセスしようとするとき，トランザクションから出ている有向枝に沿ってデッドロック検出メッセージを出し，これがもとのトランザクションに戻ってくることを調べる．

(2) 各トランザクションが有向枝をたどって到達できるすべてのトランザクション（到達可能集合）を記憶しておく．トランザクションAがトランザクションBのロックしている資源にアクセスしようとするとき，AはBの記憶している到達可能集合にA自身が含まれているか否かを確かめる．

(1)はメッセージを逐次的に送って調べなければならず効率が悪いが，(2)はアンロック時の到達可能集合の変更が容易でないという問題がある．

デッドロックの対策として時限ロック (lock timeout) を用いる方法もある．ロックに時限を付けておき，それ以後は他からアクセス要求があったときにロックを解放するというものである．ファイルサーバでよく用いられている．この方式の問題点は，負荷によってトランザクションの実行時間が変わるので，時限の設定がむずかしい点である．

⑲日本国特許庁（JP）　⑪特許出願公開

⑫公開特許公報（A）　昭62－209635

⑤Int.Cl.⁴	識別記号	庁内整理番号
G 06 F 12/00	301	6711－5B
9/46	340	F－8120－5B

⑬公開　昭和62年(1987)9月14日

審査請求　有　発明の数 1　（全4頁）

⑭発明の名称　非同期通信におけるシステム管理ファイルアクセス方式

⑳特　願　昭61－24995

㉒出　願　昭61(1986)2月7日

㉑発明者　仲川　真弘　川崎市中原区上小田中1015番地　富士通株式会社内

㉑出願人　富士通株式会社　川崎市中原区上小田中1015番地

㉔代理人　弁理士　長谷川　文廣　外1名

明　細　書

1．発明の名称

非同期通信におけるシステム管理ファイルアクセス方式

2．特許請求の範囲

通信資源等を管理するシステム管理ファイル（16）をそなえ，任意のジョブからの非同期の通信要求を随時処理する情報処理システムにおいて，

任意数の通信要求元ジョブ（11，12，13）からの依頼を受付けてキュー管理し，順次1つずつ通信要求元ジョブの依頼を取出してシステム管理ファイル（16）のアクセス処理を実行するただ1つの制御ジョブ（14）を設け，前記制御ジョブ（14）を介してシステム管理ファイル（16）をアクセスすることを特徴とする非同期通信におけるシステム管理ファイルアクセス方式。

3．発明の詳細な説明

〔概要〕

複数のジョブから通信要求があった場合，要求管理を行なうシステム管理ファイルの管理データ，たとえば資源の使用中／空き等の情報を更新するが，その際管理ファイルを更新する制御ジョブを設け，この制御ジョブのもとで，各ジョブからの要求をキュー管理させ，各要求ごとのシステム管理ファイルの更新処理を時間的に直列化することにより，ジョブ間の排他制御を不要にする。

〔産業上の利用分野〕

本発明は，情報処理システムに関するものであり，特に非同期通信のためのシステム管理ファイルアクセス方式に関する。

〔従来の技術〕

情報処理システムにおいて，ホストCPUの複数のユーザジョブが，それぞれ端末に対する非同期通信（随時実行可能な通信）を要求する場合，

各ジョブは，回線等の各通信資源ごとの使用中／空き状態その他の制御情報を管理するシステム管理ファイルにアクセスして関連する制御情報を書き替え，資源の獲得，解放を行なう。たとえば通信開始時には空き状態を使用中状態に，通信終了時には使用中状態を空き状態に変更する。

第3図は，これを概念的に図示したもので，31，32，33はそれぞれ通信要求元ジョブ，34はシステム管理ファイルである。

システム管理ファイル34は，1ボリューム内のファイルである場合が多く，通信要求をもつ複数のジョブは，I／Oネックとなるため同時並列アクセスすることはできないのが普通である。

またシステム管理ファイル34は，管理単位となる複数の管理ブロックで構成され，1つの通信要求を処理するために必要とされる制御情報は，そのうちの幾つかの管理ブロックにまたがって存在している。これは，1つの通信要求を実行するためには，関連する複数種類の通信資源が同時に使用され，それらが別々に管理されているからである。

なお，システム管理ファイル34に対するI／O処理でアクセス単位（転送単位）となる1つのファイルブロックの大きさは，通常，管理ブロックが複数個含まれる大きさである。

このようなシステム管理ファイル34を用いてジョブの通信要求を処理する場合，通信開始時の資源獲得と通信終了時の資源解放のために，関連する複数の管理ブロックをジョブごとに同時的に更新しなければならない。しかも各ジョブが発した資源獲得と資源解放のためのファイル更新要求は別々にキュー管理され，1回のファイルアクセスで済ますことができない場合が多い。このため，複数のジョブから多重に通信要求が行なわれた場合のファイル更新性を保証するため，従来はジョブ間の排他制御を行なっている。

〔発明が解決しようとする問題点〕

従来のシステムでは，システム管理ファイルに対する通信要求元ジョブによる更新処理は，通信開始時と通信終了時に行なわれ，また複数の管理ブロックが更新対象となるために，複数回のファイルアクセスが必要となることから，ジョブ間の排他制御は複雑なものとなり，またある通信要求元ジョブがシステム管理ファイルの更新処理中に異常終了した場合そのリカバリ（回復）処理が必要となるが，そのジョブが獲得していた通信資源を確実に解放させるのは極めて困難であるという問題があった。

〔問題点を解決するための手段〕

本発明は，システム管理ファイルのアクセス処理を実行できるただ1つの制御ジョブを設け，システム管理ファイルに対するアクセス要求をもつ全ての通信要求元ジョブでは，上記制御ジョブに依頼してアクセス処理を実行させるようにするものである。

第1図に本発明の原理的構成を示す。

図において，11，12，13は通信要求元ジョブで，14は制御ジョブ，15はアクセス要求キュー，16はシステム管理ファイル，17は管理ブロックである。

制御ジョブ14は，各通信要求元ジョブ11，12，13からのシステム管理ファイル16に対するアクセス要求を受け付けると，それらをアクセス要求キュー15につなぎ，先頭から順次システム管理ファイル16のアクセス処理を実行する。

制御ジョブ14は，システム管理ファイル16の管理ブロック17のうちアクセス要求キュー15から取出した1つの通信要求元ジョブに関連する全ての管理ブロック（斜線を付して示す）について，通信開始時の資源獲得と通信終了時の通信資源解放のための更新処理を実行した後，アクセス要求キュー15から次の通信要求元ジョブを取出し，実行する。

〔作用〕

本発明によれば，複数のジョブにより通信要求が同時多発しても，システム管理ファイルに対する各ジョブのアクセス処理は，単一の制御ジョブ

を介して直列化されるため，ジョブ間の排他制御は不要となり，また，ある通信要求元ジョブがアクセス処理途中で異常終了となっても，比較的簡単にリカバリ処理を行なうことができる。

〔実施例〕

次に，実施例を用いて本発明の詳細を説明する。

第２図は，本発明の１実施例の構成図である。

図において，２１，２２は通信要求元ジョブ，２３は制御ジョブ，２４は管理ブロックキュー，２５はビットマップキュー，２６はビットマップ，２７はシステム管理ファイル，２８はビットマップブロック，２９は管理ブロックを表わす。

ビットマップ２６は，システム管理ファイル２７の各管理ブロック２９と一対一に対応しており，１つの通信要求元ジョブが更新した管理ブロックに対応するビットオンに設定することにより，更新の有無の記録をとる。

システム管理ファイル２７の先頭ブロックはビットマップ２６を格納するビットマップブロック２８として使用される。

制御ジョブ２３は，通信要求元ジョブ２１，２２からの通信要求に応答して，受付け順に管理ブロックキュー２４を作成する。

制御ジョブ２３は，管理ブロックキュー２４から取出した１つの通信要求元ジョブについて，対象となる全ての管理ブロックのチェインをとって更新処理を実行する。なお，管理ブロック間のチェイン（リンク）付けは，相対ブロックナンバでリンク付けを行う。また更新対象となった各管理ブロックは，ビットマップ２６上でオン表示される。このビットマップ２６をシステム管理ファイル２７のビットマップブロック２８に書込む処理は，ビットマップキュー２５を用いて管理され，１回の１／０処理で実行される。これは，複数回に分けてビットマップの書込みを行なえば，途中でシステムダウンしたときに復元不能となるからである。システムダウンが起きた場合には，その後のシステム立上げ時に，ビットマップブロック２８を読出し，そのビットマップ中でオン表示されている管理ブロックのみを信用して制御すればよい。つまり，ビットマップがオンのファイル内の管理ブロックのみが正しく処理されたものであり，ビットマップがオフのファイル内の管理ブロックのリンク付けは，不当なものと考えればよい。このようにすることで，ファイル内で管理ブロックをリンク中にシステムダウンしても次の立上げで，仕掛り中の要求を処理してしまうことはない。

ビットマップ２６の書込みが正しく行なわれたならば，通信要求元ジョブに制御を返す。そして制御ジョブは管理ブロックキュー２４から次の要求を取出し，そのアクセス処理を実行する。

〔発明の効果〕

本発明によれば，任意複数のユーザジョブが通信要求を行なっても，制御ジョブにより全て直列処理化されるため，複雑な排他制御が不要となり，また障害発生時の資源解放処理を容易化することができ，システムの運用効率を上げることができる。

４．図面の簡単な説明

第１図は本発明の原理的構成図，第２図は本発明の１実施例の構成図，第３図は従来のシステム管理ファイルアクセス方式の概念図である。

第１図中，

１１，１２，１３：通信要求元ジョブ

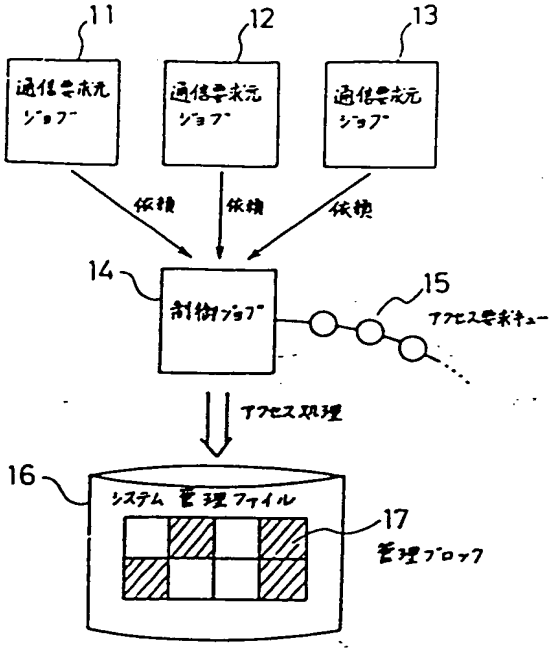
１４：制御ジョブ

１５：アクセス要求キュー

１６：システム管理ファイル

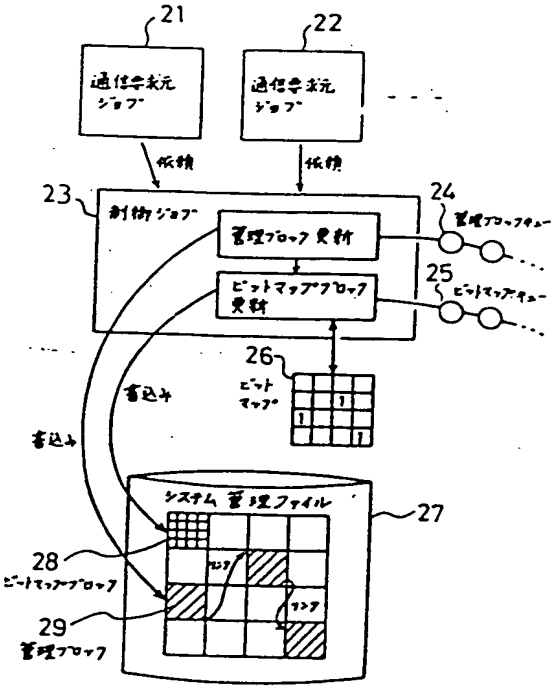
特許出願人　　富士通株式会社

代理人弁理士　　長谷川　文　廣（外１名）



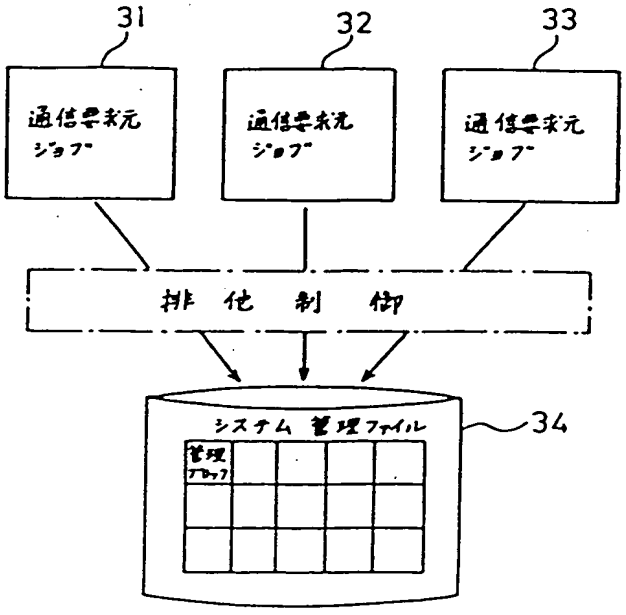
本発明の原理的構成

第 1 図



本発明の1実施例構成

第 2 図



従来のシステム管理ファイルアクセス方式

第 3 図

This Page is Inserted by IFW Indexing and Scanning Operations and is not part of the Official Record

BEST AVAILABLE IMAGES

Defective images within this document are accurate representations of the original documents submitted by the applicant.

Defects in the images include but are not limited to the items checked:

- ☒ BLACK BORDERS
- ☐ IMAGE CUT OFF AT TOP, BOTTOM OR SIDES
- ☐ FADED TEXT OR DRAWING
- ☐ BLURRED OR ILLEGIBLE TEXT OR DRAWING
- ☐ SKEWED/SLANTED IMAGES
- ☐ COLOR OR BLACK AND WHITE PHOTOGRAPHS
- ☐ GRAY SCALE DOCUMENTS
- ☐ LINES OR MARKS ON ORIGINAL DOCUMENT
- ☐ REFERENCE(S) OR EXHIBIT(S) SUBMITTED ARE POOR QUALITY
- ☐ OTHER: _____

IMAGES ARE BEST AVAILABLE COPY.
As rescanning these documents will not correct the image problems checked, please do not report these problems to the IFW Image Problem Mailbox.